PUCRS ESCOLA POLITÉCNICA

Projeto e Otimização de Algoritmos

Programação Dinâmica

MARISTA

Prof. Rafael Scopel

Programação Dinâmica

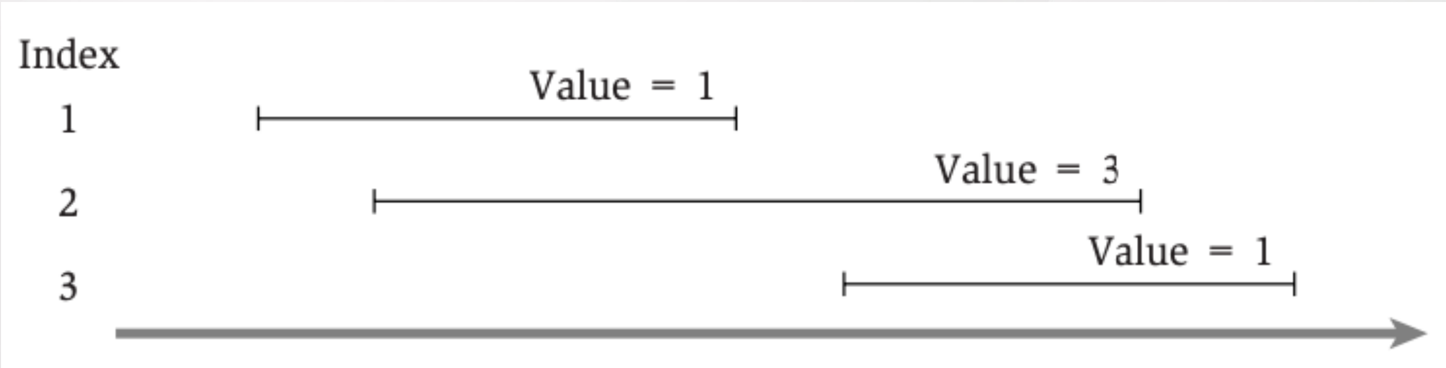
- A ideia básica para Programação Dinâmica é similar a intuição dos algoritmos de divisão e conquista, mas é essencialmente o oposto dos algoritmos greedy.
- Na programação dinâmica implicitamente procuramos o espaço de todas as soluções possíveis.
- Procedemos cuidadosamente decompondo o problema principal numa série de subproblemas e depois construímos a solução correta para subproblemas cada vez maiores.
- De certa maneira a programação dinâmica opera muito próximo da busca via força bruta.
- Apesar de estar explorando um conjunto exponencialmente grande de possíveis soluções para o problema, ela faz isso sem nunca examinar explicitamente todas as soluções.

O PROBLEMA:

- Já verificamos que um algoritmo greedy é capaz de produzir uma solução ótima para o problema de Escalonamento de Tarefas, onde o objetivo é aceitar o maior número de tarefas não sobrepostas.
- O problema de Escalonamento de Tarefas Ponderadas é uma versão geral do problema visto anteriormente, no qual cada intervalos tem um valor (ou peso) e queremos aceitar o conjunto de tarefas com o maior valor.

DESENHANDO UM ALGORITMO RECURSIVO:

- O problema original de Escalonamento de Tarefas é um caso especial do Escalonamento de Tarefas Ponderadas onde todas as tarefas tem valor 1.
- Mas o algoritmo que funcionou para resolver o problema de tarefas com valor 1, não consegue mais produzir soluções ótimas quando no cenário mais geral da figura abaixo:



- Na verdade não se tem conhecimento de nenhum algoritmo greedy capaz de resolver o Escalonamento de Tarefas Ponderadas.
- Portanto, precisamos de uma nova abordagem...a Programação Dinâmica!!!

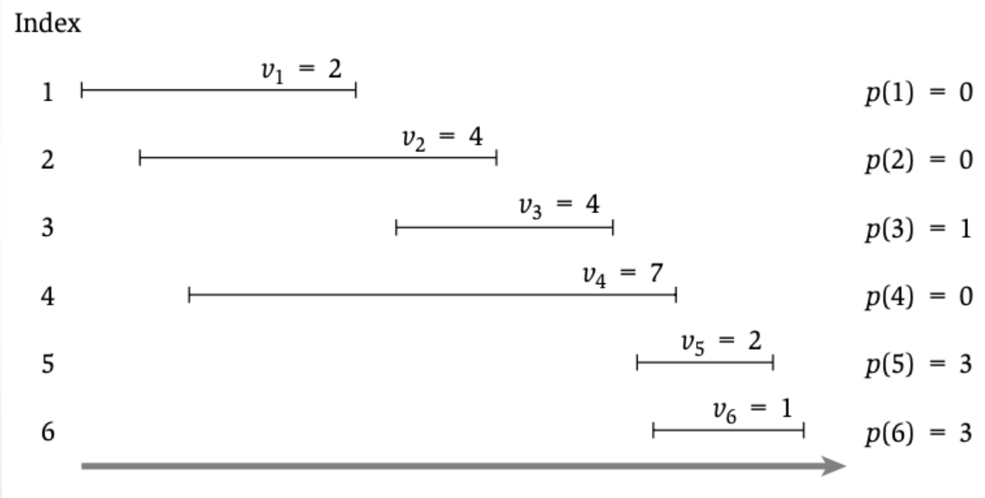
Programação Dinâmica:
Escalonamento de Tarefas Ponderadas

- Começaremos nossa introdução a Programação Dinâmica com um algoritmo recursivo para este problema.
- Temos n requisições rotuladas $1, \ldots, n$.
- Com cada requisição i especificando um tempo de início s_i e um tempo de termino f_i.
- Cada requisição agora possui um valor, ou peso, v_i.
- Dois intervalos são compatíveis se eles não se sobreporem, como antes.
- O objetivo do nosso problema é selecionar um subconjunto $S \subseteq \{1, \ldots, n\}$ de intervalos mutuamente compatíveis, de maneira a maximizar a soma dos valores de todos os intervalos:

$$\sum_{i \in S} v_i$$

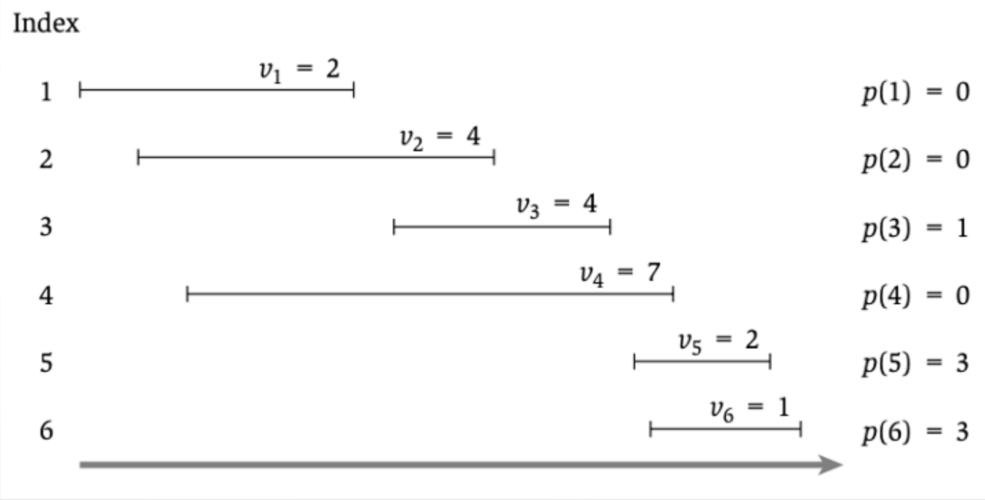
- Vamos supor que as requisições são ordenadas por tempos de finalização:

$$f_1 \leq f_2 \leq \cdots \leq f_n$$



- Dizemos que i vem antes de j se $i < j$.
- Esta será a ordem natural (esquerda para direita) que iremos considerar os intervalos.
- Definimos $p(j)$, para um intervalo j, como o maior índice $i < j$ de forma que as requisições i e j são compatíveis.
- Ou seja, i é o intervalor mais a esquerda que termina antes de j.
- Definimos $p(j) = 0$ se nenhuma requisição $i < j$ é compatível.

- Dada uma instância do problema de Escalonamento de Tarefas Ponderadas, vamos considerar uma solução ótima $\mathcal{O}$.
- Por hora vamos ignorar que não temos a menor ideia do que é $\mathcal{O}$!
- Porém, podemos dizer algo óbvio sobre $\mathcal{O}$: ou o intervalo n (o último intervalo) pertence a $\mathcal{O}$ ou ele não pertence.
- Vamos explorar essa afirmação um pouco mais.
- Se $n \in \mathcal{O}$, então claramente nenhum intervalo entre $p(n)$ e n pode pertencer a $\mathcal{O}$.
- Por causa da definição de $p(n)$, sabemos que os intervalos $p(n)+1, p(n)+2, \dots, n-1$ todos são incompatíveis com o intervalo n.
- Adicionalmente, se $n \in \mathcal{O}$, então $\mathcal{O}$ deve incluir uma solução ótima para o problema consistindo de $\{1, \dots, p(n)\}$ requisições.
- Se isso não fosse verdade, poderíamos substituir as requisições em $\mathcal{O}$, $\{1, \dots, p(n)\}$, com uma melhor, sem riscos de incompatibilidade com a requisição n.



- Porém, se $n \notin \mathcal{O}$, então $\mathcal{O}$ é simplesmente igual a solução ótima para o problema consistindo das requisições contidas no conjunto $\{1, \dots, n-1\}$.
- Isso ocorre por raciocínio análogo: assumimos que $\mathcal{O}$ não inclui a requisição n; então se ele não possuir uma solução dentro do conjunto de soluções $\{1, \dots, n-1\}$, poríamos encontrar uma melhor.
- Tudo isso indica que para encontrar a solução em intervalos $\{1,2, \dots, n\}$ precisamos procurar pela solução ótima em problemas menores da forma $\{1,2, \dots, j\}$.

PUCRS ESCOLA POLITÉCNICA

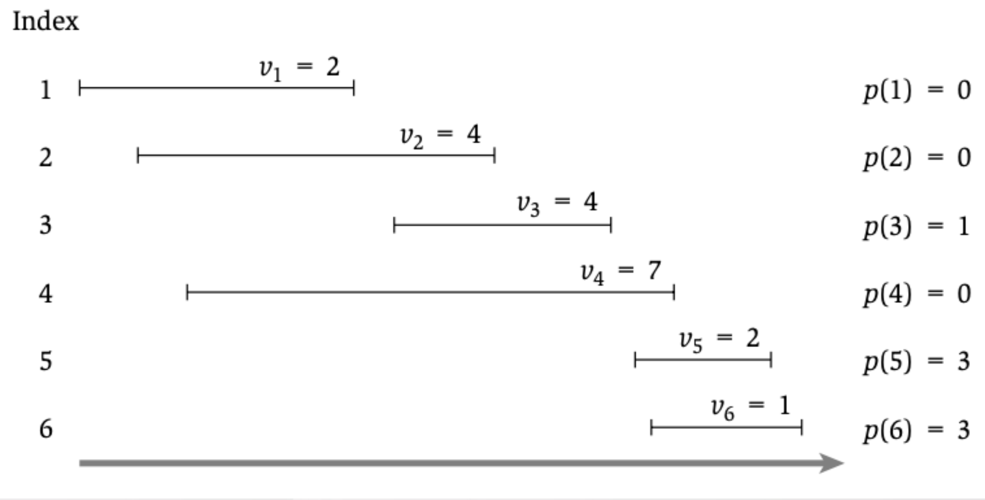
Programação Dinâmica:
Escalonamento de Tarefas Ponderadas

- Para qualquer valor de j entre 1 e n, deixe $\mathcal{O}_j$ denotar a solução ótima para o problema consistindo de requisições $\{1,2,\ldots,j\}$ e deixe $\mathrm{OPT}(j)$ definir o valor dessa solução.
- Definimos $\mathrm{OPT}(0) = 0$, baseado na convenção de que é um ótimo sobre um conjunto vazio de intervalos.
- A solução ótima que estamos procurando é precisamente $\mathcal{O}_n$, com o valor $\mathrm{OPT}(n)$.
- Para a solução ótima $\mathcal{O}_j$ em $\{1,2,\ldots,j\}$, nossa abordagem (generalizando do caso em que $j = n$) afirma que ou $j \in \mathcal{O}_j$ e nesse caso:

$$\mathrm{OPT}(j) = v_j + \mathrm{OPT}\big(p(j)\big)$$

Ou $j \notin \mathcal{O}_j$ e nesse caso:

$$\mathrm{OPT}(j) = \mathrm{OPT}(j-1)$$



- Como essas são as duas únicas possibilidades temos:

A.20:

$$\mathbf{OPT}(j) = \begin{cases} \mathbf{0} & , se\ j = 0 \\ max\left(v_j + \mathbf{OPT}\big(p(j)\big), \mathbf{OPT}(j-1)\right) & , se\ j > 0 \end{cases}$$

Programação Dinâmica:
Escalonamento de Tarefas Ponderadas

A.20:

$$\mathbf{OPT}(j) = \begin{cases} \mathbf{0} & \boldsymbol{, se\ j = 0} \\ \boldsymbol{max}\left(\boldsymbol{v_j} + \mathbf{OPT}(\boldsymbol{p}(\boldsymbol{j})), \mathbf{OPT}(\boldsymbol{j} - \mathbf{1})\right) & \boldsymbol{, se\ j > 0} \end{cases}$$

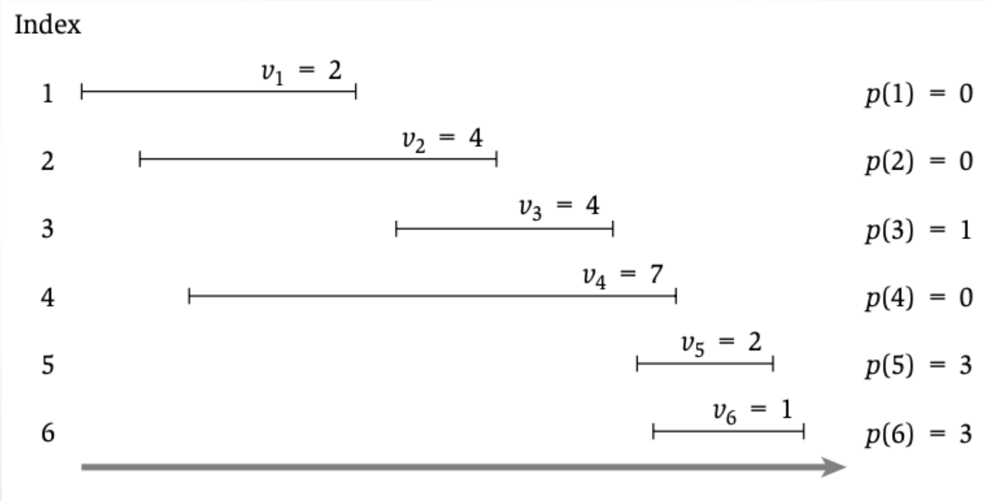
- Como definimos que n pertence a solução ótima $\mathcal{O}_j$?
- Fácil: n pertence a solução ótima se e somente se a primeira das opções acima é no mínimo tão boa quanto a segunda, em outras palavras

A.21: A requisição j pertence a uma solução ótimas no conjunto $\{1, 2, \dots, j\}$ se e somente se:

$$\boldsymbol{v_j} + \mathbf{OPT}(\boldsymbol{p}(\boldsymbol{j})) \geq \mathbf{OPT}(\boldsymbol{j} - \mathbf{1})$$

- Esses fatos formam o principal componente no qual uma solução de programação dinâmica é baseada:

Uma equação de recorrência que expressa a solução ótima (ou o seu valor) em termos de soluções ótimas para subproblemas menores



- A.20 nos fornece um algoritmo para computar $\mathrm{OPT}(n)$, assumindo que:
  - já ordenamos as requisições pelo tempo de finalização
  - Computamos os valores de $p(j)$ para cada j.

```
Compute-Opt(j)
  If j=0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), Compute-Opt(j − 1))
  Endif
```

Programação Dinâmica:
Escalonamento de Tarefas Ponderadas

```
Compute-Opt(j)
  If j=0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), Compute-Opt(j − 1))
  Endif
```

A.22: Compute-Opt(j) **computa corretamente** $\mathbf{OPT}(\boldsymbol{j})$ **para cada** $\boldsymbol{j} = \mathbf{1}, \mathbf{2}, \dots, \boldsymbol{n}$.

Prova:

- Por definição $\mathrm{OPT}(0) = 0$.
- Agora, assuma um $j > 0$, e suponha por indução que Compute-Opt(i) para todo o $i < j$.
- Pela hipótese de indução, sabemos que

$$\text{Compute-Opt}(p(j)) = \mathrm{OPT}\big(p(j)\big)$$

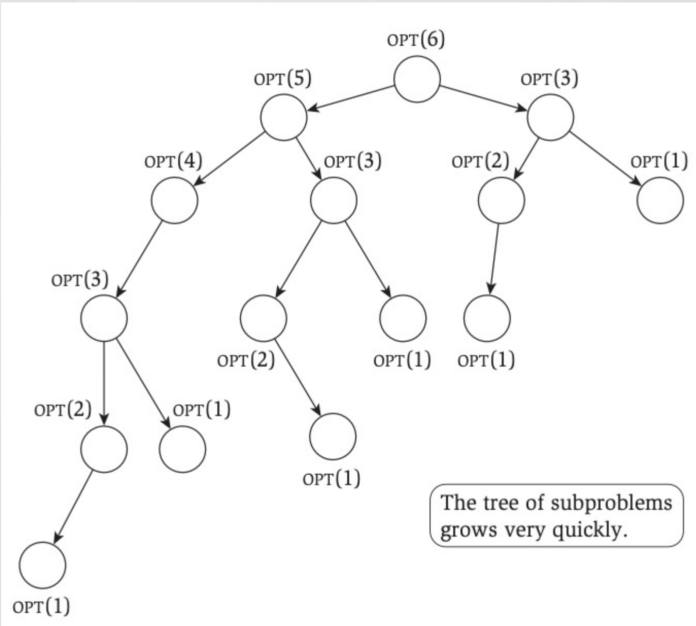
e que

$$\text{Compute-Opt}(j-1) = \mathrm{OPT}(j-1).$$

- Por A.20 temos que:

$$\begin{aligned}\mathrm{OPT}(j) = \ &\mathbf{max}\big(\boldsymbol{v_j} + \text{Compute–Opt}(p(j)), \text{Compute–Opt}(j-1)\big) \\ &= \text{Compute–Opt}(j),\end{aligned}$$

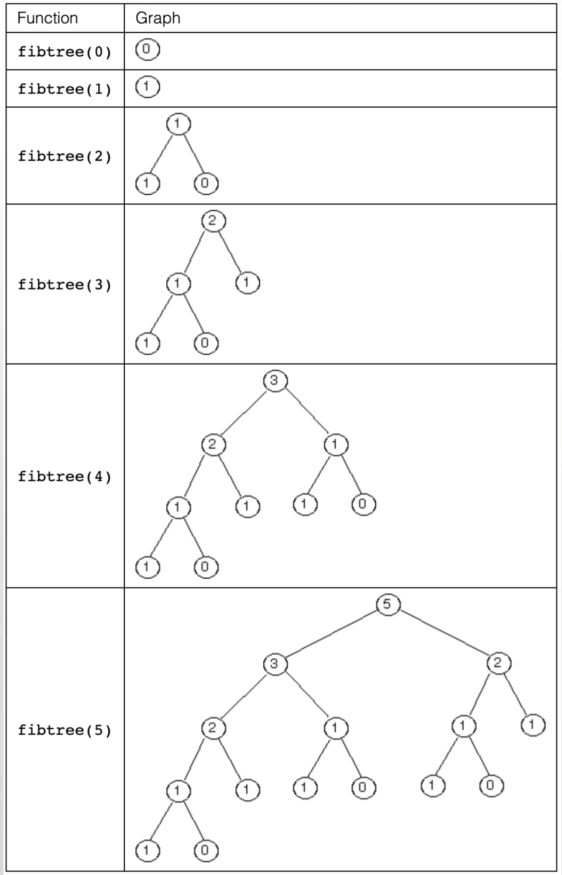
- **Prova concluída.**
- Infelizmente, se implementarmos o algoritmo descrito por Compute–Opt ele levaria tempo exponencial para rodar no pior caso.
- A árvore abaixo demostra a velocidade com que a árvore de recursão cresce para o nosso exemplo inicial:

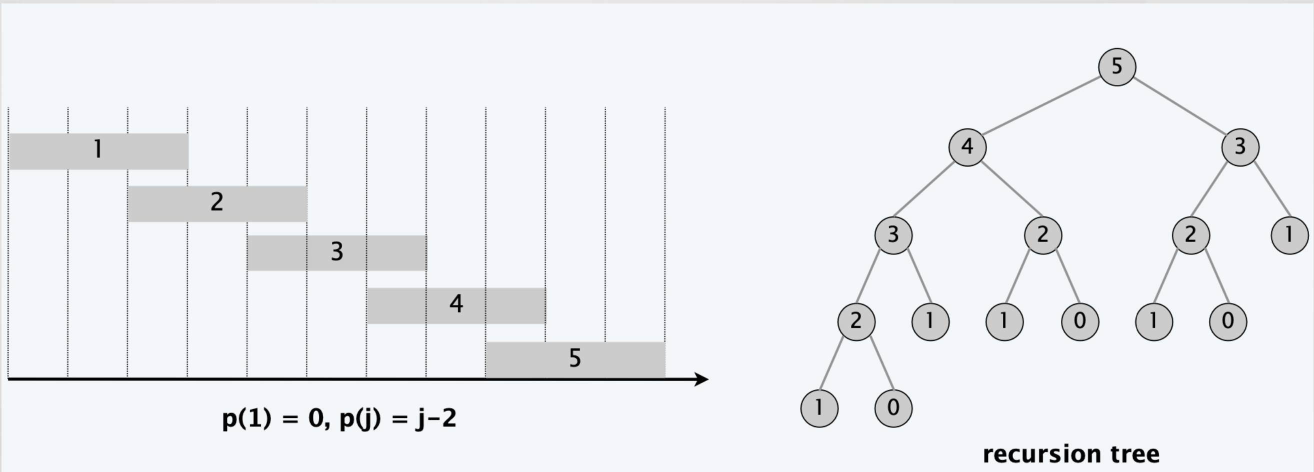


Programação Dinâmica:
Escalonamento de Tarefas Ponderadas

```
Compute-Opt(j)
  If j=0 then
    Return 0
  Else
    Return max(v_j+Compute-Opt(p(j)), Compute-Opt(j − 1))
  Endif
```

- Um caso extremo é o da figura ao lado, onde as requisições forma uma camada de instâncias empilhadas no formato de escada.
- Onde $p(j) = j - 2$ para cada $j = 2,3,4, \ldots, n$, podemos verificar que Compute-Opt(j) invoca chamadas recursivas de tamanho $j - 1$ e $j - 2$.
- Fazendo com que o problema cresça como uma árvore da sequência de Fibonacci e, portanto, exponencialmente.
- E isso nos leva para longe de uma solução desejada em tempo polinomial.



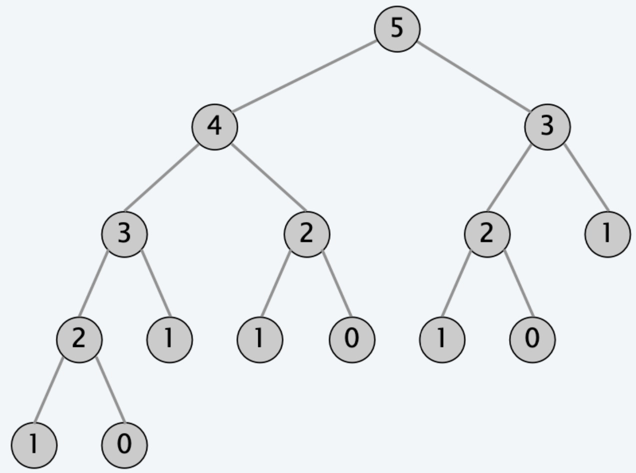


MEMOIZANDO A RECURSÃO:

- Porém, em termos algorítmicos, não estamos tão distantes de obter uma solução em tempo polinomial.
- Um observação fundamental, e que forma o segundo componente crucial de uma solução de programação dinâmica, é que o algoritmo recursivo Compute-Opt está somente resolvendo $n + 1$ diferentes subproblemas: Compute-Opt(0), Compute-Opt(1),..., Compute-Opt(n).
- O fato que ele executa em tempo exponencial é simplesmente devido a espetacular redundância no número de vezes que ele realiza cada uma dessas chamadas. Por exemplo, na árvore do problema chamamos

  Compute-Opt(3) 2 vezes, Compute-Opt(2) 3 vezes e

  Compute-Opt(1) 5 vezes.
- Como poderíamos eliminar todas essa redundância?



recursion tree

Programação Dinâmica:
Escalonamento de Tarefas Ponderadas

- Poderíamos armazenar o valor de Compute-Opt numa variável global na primeira vez que computarmos o valor e depois continuar a utilizar o valor pré-computado nas chamadas recursivas subsequentes.
- Essa técnica de salvar valores que já foram computados é chamada de **MEMOIZAÇÃO**!
- Vamos ajustar o algoritmo do Compute-Opt para permitir a memorização, chamado M-Compute-Opt.
- O novo algoritmo faz uso de um vetor $M[0 \ldots n]$.
- $M[j]$ começa com o valor inicial vazio, mas será atualizado assim que Compute-Opt(j) for computado.
- Para determinar $\mathrm{OPT}(n)$ invocamos M-Compute-Opt(n).

***A.23:* O tempo de execução de M-Compute-Opt(n) é $O(n)$, assumindo que os intervalos foram ordenados pelos seus tempos de finalização**

```
M-Compute-Opt(j)
  If j = 0 then
    Return 0
  Else if M[j] is not empty then
    Return M[j]
  Else
   Define M[j] = max(v_j+M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
    Return M[j]
 Endif
```

A.23: **O tempo de execução de** $\text{M-Compute-Opt}(n)$ **é** $O(n)$**, assumindo que os intervalos foram ordenados pelos seus tempos de finalização**

Prova

- O tempo para computar uma única chamada de M-Compute-Opt é $O(1)$, excluindo-se o tempo gasto nas chamadas recursivas que ele realiza.
- Portanto, o tempo de execução é limitado por uma constante vezes o numero de todas as chamadas de M-Compute-Opt.
- Como a implementação não define um limite superior explícito no número de chamadas, tentaremos encontrar um limite olhando para uma boa medida de progresso.
- A medida de progresso mais útil aqui é o número de entradas em M que não são vazios.

```
M-Compute-Opt(j)
  If j=0 then
    Return 0
  Else if M[j] is not empty then
    Return M[j]
  Else
    Define M[j] = max(v_j+M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
    Return M[j]
  Endif
```

- Inicialmente este número é zero, mas a cada vez que o algoritmo invoca a recorrência, gerando duas novas chamadas de M-Compute-Opt, uma nova entrada é preenchida.
- Isto incrementa o número de entradas preenchidas em 1.
- Como M tem apenas $n + 1$ entradas, temos que podem ocorrer no máximo $O(n)$ chamadas para M-Compute-Opt.
- Portanto, o tempo de execução de M-Compute-Opt é $O(n)$ como desejado.
- **Prova concluída.**

Computando o conjunto de Solução:

- Até o momento nós computamos o valor da solução ótima, porém, também desejamos encontrar o conjunto S de requisições que compõem a solução ótima.
- Seria fácil ajustar M-Compute-Opt para manter um registro da solução ótima em adição ao valor da solução.
- Manteríamos um vetor S de forma que $S[i]$ contem o conjunto ótimo de intervalos entre $\{1,2,3,\dots,i\}$.
- Ingenuamente aumentando o código para manter as soluções no vetor S, entretanto, iria explodir o tempo de execução por um fator adicional de $O(n)$.
- Enquanto atualizar uma posição em M leva $O(1)$, escrever um conjunto no vetor S leva $O(n)$.
- Podemos evitar essa explosão de $O(n)$ não mantemos explicitamente S, mas sim recuperamos a solução ótima dos valores salvos em M após um valor ótimo ter sido computado.

- Sabemos por A.21 que j pertence a solução ótima para o conjunto de intervalos $\{1,2,3,\dots,j\}$ se e somente se:

$$v_j + OPT\big(p(j)\big) \geq OPT(j-1)$$

- Usando essa observação, obtemos o seguinte simples procedimento, que percorre na ordem inversa o vetor M para encontrar o conjunto de intervalos de uma solução ótima.

```
Find-Solution(j)
  If j=0 then
    Output nothing
  Else
    If v_j + M[p(j)] ≥ M[j−1] then
      Output j together with the result of Find-Solution(p(j))
    Else
      Output the result of Find-Solution(j−1)
    Endif
  Endif
```

- Como Find-Solution é chamado recursivamente apenas em valores menores, ele faz um total de $O(n)$ chamadas recursivas, e como gasta um tempo constante por chamada temos:

A.24: **Data um vetor M de valores ótimos para os subproblemas Find-Solution retorna uma solução ótima em tempo $O(n)$.**